Étude Préliminaire À Une Utilisation De Mémoires Secondaires Pour Le Stockage Des Métadonnées Java Dans Des Systèmes Contraints.

Geoffroy Cogniaux, Michaël Hauspie, François-Xavier Marseille

RenPar'20 / SympA'14 / **CFSE 8**
Saint-Malo, France, du 10 au 13 mai 2011

Étude préliminaire à une utilisation de mémoires secondaires pour le stockage des métadonnées java dans des systèmes contraints

Geoffroy Cogniaux[†‡] Michaël Hauspie[‡] Francois-Xavier Marseille[†]

† Gemalto Technology & Innovation, France *{prenom.nom}@gemalto.com*
‡ LIFL, CNRS UMR 8022, Univ. Lille 1, INRIA Lille - Nord Europe, France *{prenom.nom}@lifl.fr*

Résumé
L'utilisation de langages robustes et de haut niveau, basés sur des machines virtuelles comme le Java, est désormais possible dans des systèmes aussi petits que les cartes à puce ou les capteurs réseau. Toutefois, le potentiel de ces langages est encore largement bridé par les ressources physiques de ces systèmes, notamment l'espace dédié au stockage du code et des données de fonctionnement de tels langages (*i.e.* les métadonnées). Nous pensons qu'il est possible de lever ce verrou en couplant des mécanismes de cache avec la présence de plus en plus fréquente dans ces systèmes de mémoires secondaires externes et volumineuses. Ce papier est une étude préliminaire à la mise en œuvre de cette nouvelle approche. Grâce à des résultats obtenus par simulation, nous avons identifié trois facteurs qui concourent à dégrader les performances de la mise en cache de code Java et de ses métadonnées : le niveau d'abstraction objet des applications, la fragmentation des métadonnées et le phénomène de dilution spatiale occasionnée par le cache sur ces données fragmentées et très petites. Au vue de nos analyses, nous pensons qu'une solution ne peut pas être idéale sans être une réponse globale à ces trois facteurs.

Mots-clés : Java, Métadonnées, Cache, Mémoires secondaires, Systèmes contraints

1. Introduction

Les langages de haut niveau orientés objet comme le langage Java [2] facilitent et accélèrent le développement d'applications plus sûres, plus robustes et réutilisables. Ils semblent donc être également de bons choix pour les systèmes embarqués qui nécessitent ce type d'attention vis à vis des applications qu'ils embarquent. Néanmoins, la programmation orientée objet repose généralement sur des besoins plus importants en terme de mémoire. Malheureusement, dans les systèmes fortement contraints, la mémoire est la ressource physique la plus critique car elle est un facteur de coût de production très important et est donc disponible en faible quantité.

Dans la plus basse catégorie des systèmes embarqués, celle des petits objets intelligents (ou SmartObject, comme les cartes à puce, les capteurs réseaux, les micro-contrôleurs...), la mémoire interne est si rare qu'elle n'excède que rarement les 32Ko de mémoire volatile (RAM) et 1Mo de mémoire non-volatile, comme la Flash NOR ou l'EEPROM. Par contre, ces systèmes possèdent de plus en plus souvent des mémoires non-volatiles externes, c'est à dire en dehors de l'espace d'adressage du CPU et accessibles à travers un bus, ce qui rend inévitablement plus lents les temps d'accès. Cette mémoire secondaire peut-être de la NOR série (entre 8Mo et 64Mo) ou de la Flash NAND qui peut compter plusieurs Go sous forme de cartes MicroSD par exemple dans certains capteurs.

Depuis de nombreuses années, la réduction de la consommation mémoire a été l'une des questions les plus étudiées dans la communauté des machines virtuelles pour micro-contrôleurs, et a toujours débouché sur des compromis, parfois très radicaux, par rapport aux spécifications originelles de Java. Si nous regardons l'histoire de ce langage dans le contexte de ces systèmes, on peut constater une radicalisation graduelle, aboutissant à des sous-ensembles de spécifications de plus en plus pauvres par rapport aux promesses du langage. Si J2SE (Java 2 Standard Edition) est toujours la référence, on trouve aujourd'hui des spécifications adaptées autant que possible à la taille des systèmes cibles : J2ME (Java 2 Micro Edition) CDC (Connected Device Configuration) pour les processeurs 32-bits avec au moins 2 Mo de mémoire ; J2ME CLDC (Connected Limited Device Configuration) pour les processeurs 16-bits avec au

moins 160 Ko ; ou encore Java-Card pour les cartes à puce dotées de processeurs 8/16-bits avec au moins 4 Ko de RAM, 16 Ko de ROM et de 8 à 128 Ko de mémoire flash.
Dans l'informatique traditionnelle, le problème de contrainte mémoire a été résolu par l'utilisation de mécanismes de virtualisation de la mémoire reposant sur l'utilisation d'une partition d'échange située sur une mémoire secondaire, généralement un disque dur. Malheureusement, l'absence de support matériel adapté rend très difficile son utilisation dans les micro-contrôleurs. L'utilisation d'une machine virtuelle (VM) peut permettre de contourner plus facilement ce problème en contrôlant les accès mémoire et gérant elle même la pagination à la demande (*i.e.* accéder à la mémoire secondaire uniquement lorsque cela est nécessaire).
Les données de fonctionnement (ou métadonnées) sont au centre des langages interprétés car elles décrivent le comportement que la machine virtuelle doit adopter pour exécuter un programme en respectant les spécifications du langage. À l'opposé des réponses apportées jusqu'à présent, nous imaginons une alternative qui consiste à déporter la volumétrie des métadonnées de la machine virtuelle Java vers une mémoire secondaire. On peut ainsi libérer jusqu'à 60% de l'espace de travail de la machine virtuelle. De plus, les métadonnées sont quasiment exclusivement accédées en lecture, ce qui limite l'impact sur les performances : les mémoires secondaires qui nous avons à disposition étant beaucoup plus lentes en écriture qu'en lecture. Les perspectives de notre approche sont prometteuses car elle permet d'inverser la tendance à la spécialisation, et finalement la limitation, des machines virtuelles Java pour les systèmes enfouis.
Dans ce papier, nous contribuons à ces nouvelles perspectives par une série d'observations et d'analyses fondamentales, préliminaires à de futures propositions. Suite à cette analyse, nous pouvons dire que trois facteurs majeurs viennent pour l'instant perturber les performances : le modèle de programmation objet en tant que tel, la fragmentation des métadonnées et la dilution spatiale occasionnée par la mise cache de ces métadonnées fragmentées. Ces trois points doivent alors être le point de départ à la mise en œuvre d'une solution efficace.
Le reste du papier est organisé de la façon suivante : la Section 2 place dans son contexte notre contribution puis la Section 3 s'attache à décrire les concepts utilisés par la suite. Sur cette base, la Section 4 développe notre méthodologie d'analyse. À partir de nos résultats issus d'analyses de traces, la Section 5 formule une série d'observations et leur donne une explication. La section 6 présente les perpectives offertes par notre étude. Enfin, nous donnons nos conclusions en Section 7.

2. Contexte et travaux connexes

Cette Section situe notre contribution par rapport aux travaux de recherche en matière de machines virtuelles Java embarquées face à la problématique des contraintes mémoires, suivi d'une brève revue technologique des mémoires secondaires disponibles dans les systèmes enfouis d'aujourd'hui.

2.1. Contexte de l'étude

Encore aujourd'hui, la plupart des applications pour SmartObject sont écrites dans des langages de bas niveau comme le C ou l'assembleur. Pourtant, cette catégorie de systèmes est peut être celle qui doit faire face à la plus grande hétérogénéité des plate-formes, ce qui conduit inévitablement à réécrire des systèmes dédiés, des couches les plus basses comme les pilotes de périphériques (connus pour être la principale source de bogues) aux couches les plus hautes que sont les applications.
L'utilisation de machines virtuelles, y compris dans ces systèmes très contraints, est donc devenue un thème de recherche incontournable pour nombre de bénéfices reconnus par la communauté [12, 17, 5, 14]. Tout d'abord, parce qu'une machine virtuelle est une couche d'abstraction du matériel qui permet la portabilité et qui dépasse donc le problème de l'hétérogénéité des plate-formes. Ensuite, parce qu'elle permet l'utilisation de langages de plus haut niveau qui introduisent plus de robustesse pour le système. Par exemple, un gestionnaire d'exceptions permet d'isoler et de gérer des erreurs sans mettre en faillite le système complet. Enfin, des langages comme le langage Java offre un confort de développement non-négligeable de part sa sémantique *orientée objet*, la mise à disposition de composants éprouvés et prêts à l'emploi, ou la gestion automatique de la mémoire.

2.2. Travaux connexes sur les machines virtuelles Java embarquées

Tous les travaux de recherches sur la conception de machines virtuelles Java (JVM) pour systèmes embarquées fortement contraints partent du même constat : le principal obstacle pour ce genre de cibles est leur faible quantité de mémoire. Nous allons voir ici les réponses apportées à cette problématique.

L'introduction de JavaCard a été un tournant majeur dans la miniaturisation des JVM. La plateforme JavaCard est en effet la première à inscrire dans ces spécifications l'abandon du fichier de classes standard (jusque JavaCard3) au profit d'un format de fichier plus compact et d'un jeu d'instructions modifié tenant compte des spécificités des systèmes cibles. JavaCard repose sur un convertisseur s'exécutant avant le déploiement des applications sur les cartes à puce. Ce convertisseur effectue les tâches traditionnellement dévolues à un chargeur dynamique de classes : vérification des spécifications du langage au niveau du code compilé, résolutions des références symboliques, ou encore l'initialisation des variables statiques. Ce convertisseur se sert également des informations collectées pour optimiser le code et notamment distinguer et surtout dissocier les opérations 8 et 16 bits des opérations 32bits. Enfin, le convertisseur formate un fichier en sortie qui correspond quasiment au format des structures de données utilisées par la JVM à l'intérieur d'une carte JavaCard.

Squawk [17] est une des premières machine virtuelle Java à avoir réutilisé ce concept sur des fichiers de classes Java standards. Squawk part de l'hypothèse, comme JavaCard, d'un monde fermé (*i.e.* sans chargement dynamique d'applications une fois le système démarré). La nouveauté tient ici dans la conception de la JVM qui est écrite au départ en Java, puis pour par la suite, elle aussi, optimisé hors-ligne. Ce modéle permet de réintégrer plus de fonctionalités notamment le multi-threading. Mais la version 1.1 de Squawk (2006) représente encore 149Ko pour le binaire et au minimum 512Ko de code Java.

D'autres projets comme [5, 3, 6, 16] ont repoussé encore un peu plus cette limite en réduisant les fonctionnalités de la JVM aux spécifités des cibles et en ayant une approche encore plus agressive sur ce qui pouvait être effectué hors-ligne : l'abandon de la contraignante pile Java 32bits ou l'utilisation d'un autre format de *bytecode* intermédiaire, plus compact que le *bytecode* Java et spécifique à la VM. Une autre alternative comme le projet JITS (Java-In-The-Small) [7] a montré qu'il était encore possible de rester sur les standards et spécifications de Java2SE en analysant de manière abstraite et hors-ligne le code Java pour enlever toutes les fonctionalités et informations inutiles aux applications à déployer sur le système en-ligne.

Notre approche est sensiblement différente des solutions radicales et aggressives proposées jusqu'à présents. Nous imaginons une alternative qui consiste à déporter la volumétrie des données de fonctionnement (*i.e.* les métadonnées, que nous décrirons par la suite) de la machine virtuelle Java vers une mémoire secondaire, avec deux objectifs principaux : libérer de l'espace dans la mémoire principale soit pour intégrer plus de fonctionnalités à la machine virtuelle, soit pour réduire la taille la mémoire principale ; utiliser le large espace disponible de la mémoire secondaire pour embarquer beaucoup plus d'applications Java et des applications plus riches.

2.3. Hypothèse des mémoires externes

Dans les systèmes traditionnels, ce que l'on appelle la « mémoire virtuelle » est largement, et depuis longtemps, utilisée pour dépasser les problèmes de contrainte mémoire. La « mémoire virtuelle » donne l'illusion d'un espace mémoire RAM quasiment illimité en copiant des données peu utiles à un instant T sur un espace de stockage secondaire, laissant ainsi la place requise en mémoire vive. Ce mécanisme est transparent pour les applications car, en grande partie, pris en charge au niveau matériel par la MMU (Memory Management Unit), le reste étant géré par le système d'exploitation. Cependant, les matériels cibles de notre étude sont dépourvus de MMU et donc très peu de systèmes d'exploitation pour *SmartObject* [1, 8, 10] fournissent un service de « mémoire virtuelle ». Plus récemment toutefois, t-kernel [9] a démontré que ce service pouvait être envisagé dans le cadre des capteurs réseaux.

Dans le contexte des *SmartObject*, les mémoires de stockage secondaires, externes à la puce du processeur et jugées trop lentes pour d'autres utilisations, ont longtemps été considérées comme des espaces de log, gérés par des systèmes de fichiers souvent lourds et complexes. Dans [11], Lachenmann *et al.* ont cependant contribué à faire avancer l'idée d'utiliser ces mémoires externes pour étendre la quantité de mémoire de travail disponible. Ils proposent ainsi une mémoire virtuelle conçue pour TinyOS. Dans leur solution, une variable peu être étiquetée au niveau du code source comme susceptible d'être stockée provisoirement en mémoire externe, si besoin. Toutefois, cette solution repose exclusivement

sur l'hypothèse que les adresses de toutes les variables soient connues dès la compilation et qu'elles ne changent plus à l'exécution, ce qui est une spécificité que l'on ne retrouve que dans TinyOS.
TODO ? :*Dans [15], Park* et al. *ont de leurs côtés émis l'idée d'utiliser des mémoires externes pour stocker et exécuter du code compilé directement depuis cet emplacement (Exécution en place ou Execute-In-Place) à travers un cache. Dans leur solution, les appels au cache sont ajoutés au binaire du futur exécutable à la compilation. Chaque opération assembleur LOAD/STORE y est remplacée par des appels aux routines de gestion de cache, ce qui ajoute un surcoût non négligeable lorsque l'accès à celui-ci n'est pas nécessaire. De plus, la taille de l'espace RAM allouée au cache doit être conséquente (jusqu'à plusieurs centaines de kilo-octets) pour espérer des performances suffisantes.*
Ces deux propositions sont difficilement transposables pour d'autres contextes que les leurs. Cependant, elles jettent les bases de nouvelles perspectives pour l'exécution de code Java dans des systèmes enfouis en montrant que des techniques de pagination à la demande - issues des techniques bien de connues de cache et de mémoire virtuelle - sont des pistes à envisager.

2.4. Contexte technologique des petites mémoires non-volatiles

L'externalisation efficace de données d'exécution est tributaire de la technologie utilisée pour leur stockage. Il existe en effet plusieurs types de mémoires non-volatiles aux propriétés et mode d'accès très différents. Par exemple, une Flash interne typique pour microcontrôleurs peut fournir un octet en lecture toutes les 40ns à 33MHz, soit un débit crête en lecture de 23,84Mo/s. Une Flash externe, qu'elle soit de la NOR ou de la NAND, est accessible par page et non plus par octet. À la même cadence, elle doit d'abord pré-charger la page accédée (de 15µs à 25µs selon la taille d'une page), puis fournira séquentiellement un octet toutes les 50ns, soit 25,6µs pour finalement lire le dernier octet d'une page de 512 octets. La pénalité à subir par un défaut de cache peut donc être un facteur de contre-performance important avec ce type de mémoire. Lorsqu'il s'agit d'écriture, ces mémoires sont encore plus lentes car on parle ici en millisecondes. Il faut donc éviter au maximum les écritures ou choisir de ne stocker dans la mémoire secondaire que des données accédées en lecture seules.

3. Le Concept de métadonnée

3.1. Machine virtuelle Java, donnée, code et métadonnée

Comme décrit dans [13], la machine virtuelle Java (JVM) est la pierre angulaire du langage de programmation Java [2]. Une machine virtuelle est un logiciel émulant le comportement d'une machine réelle. Cette couche d'abstraction donne la vision d'un système unique et générique aux applications qu'elle exécute, et donc également aux développeurs de ces applications. La JVM est ainsi le composant clé de la plateforme Java car c'est elle qui garantit la portabilité du code compilé.
La mise en œuvre interne d'une JVM est libre et pourrait ne rien connaître du langage Java en tant que tel. L'interopérabilité entre ce langage et une JVM quelconque est assurée par le respect par celle-ci du format de fichier d'échange Java : le fichier *class*. Un fichier *class* contient le code compilé (ou *bytecodes*) d'une classe Java, une table de symboles (ou *constant pool*), et toutes les informations utiles et nécessaires pour que la JVM puisse créer et contrôler des objets Java à partir de cette classe (champs, méthodes, attributs, gestionnaire d'exceptions...). Si un objet Java est une donnée volatile qui n'existe qu'à l'exécution, le contenu d'un fichier *class* est un ensemble de métadonnées décrivant le comportement, les propriétés et les capacités de cet objet Java à l'exécution.
Les métadonnées, une fois absorbées par la JVM lors du processus de chargement de classes, peuvent être organisées en mémoire de multiples façons et cette organisation reste à la discrétion du concepteur de la machine virtuelle, ce qui en complique l'analyse. Toutefois, il est possible de dégager une vision générale de ce que sont les métadonnées Java à la lecture de ses spécifications (Figure 1).

3.2. Régions mémoire d'une machine virtuelle Java

Dans [13], les données et objets Java sont placés dans une région mémoire appelée le tas (ou *heap*), et les métadonnées dans une autre région distincte appelée *method area*, bien que, encore une fois, ce ne soit pas obligatoire. Cette dichotomie vient principalement du caractère très volatile des objets Java par opposition aux métadonnées qui sont, elles, très peu modifiées. A noter, que dans le cas de JVM classiques, ces deux espaces ne sont qu'un découpage logique et résident tous deux en mémoire vive

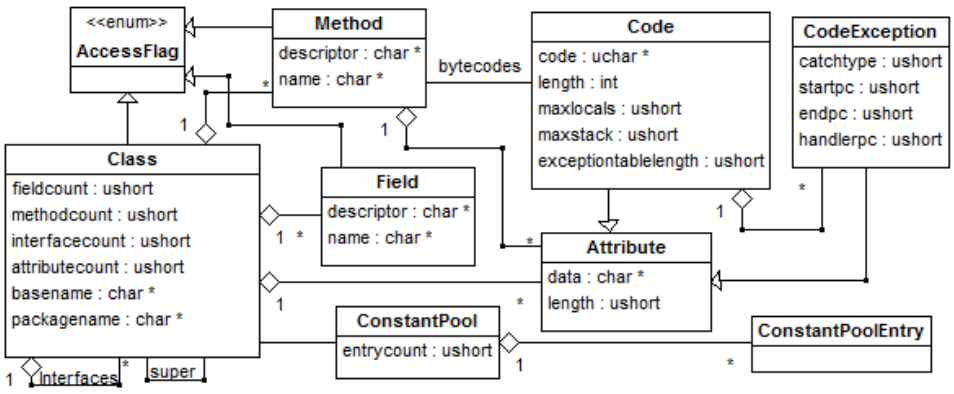


FIGURE 1 – Une représentation UML des métadonnées Java non-optionnelles.

(RAM).
Cette dichotomie est toutefois très intéressantes dans notre contexte. Si l'on considère les temps d'écriture en mémoire secondaire, il ne semble pas très judicieux d'y stocker des données volatiles. Au contraire, les métadonnnées sont quant à elles très stables dans le temps et n'obligent des écritures en mémoire secondaire que lors du processus de chargement de classes ou par l'emploi de techniques d'optimisation in-situ, elles aussi peu fréquentes.
Dans les langages interprétés comme le Java, les métadonnées sont également une source d'occupation mémoire non-négligeable. En effet, déployer plusieurs applications Java sur un capteur ou une JavaCard ne change pas la taille de la JVM. Par contre, chaque application Java apporte son lot de métadonnées supplémentaire, sachant que l'API de base J2ME représente par exemple 130Ko de métadonnées.

3.3. Synthèse

Les métadonnées sont au centre des langages interprétés car elles décrivent le comportement que la machine virtuelle doit adopter pour exécuter un programme en respectant les spécifications du langage. Elles sont aussi le point critique à étudier pour augmenter le volume d'applications Java à embarquer car elles sont volumineuse par rapport aux faibles quantités de mémoire disponible. Une solution séduisante, pour effectivement résoudre la problèmatique de contrainte mémoire, consiste à déporter la volumétrie des métadonnées vers une mémoire secondaire. La suite de ce papier propose une analyse de ce nouveau challenge.

4. Méthodologie

4.1. Hypothèse

Par ce papier, nous proposons une analyse détaillée des accès aux métadonnées dans le contexte des architectures mémoires hétérogènes des systèmes enfouis, en prenant l'hypothèse supplémentaire que cette région mémoire réside dans une mémoire dite externe, c'est à dire en dehors de l'espace d'adressage du processeur. Cette hypothèse rejoint deux spécificités liées aux métadonnées Java :
– les métadonnées Java sont accédées la plupart du temps en lecture seule ;
– l'espace occupé par les métadonnées augmente avec le nombre et la taille des applications Java.

4.2. Protocole expérimental

Mesurer l'impact du stockage des métadonnées dans une mémoire externe, qui devront être accédées à travers un cache, nécessite donc d'analyser :
– les types de métadonnée accédées ;
– la fréquence d'accès aux métadonnées ;
– l'impact de l'utilisation d'un cache.
La méthodologie suivie pour l'étude présentée ici est donc délibérément exclusivement expérimentale et se fonde sur la collecte puis l'analyse d'une quantité de données la plus grande et la plus exhaustive

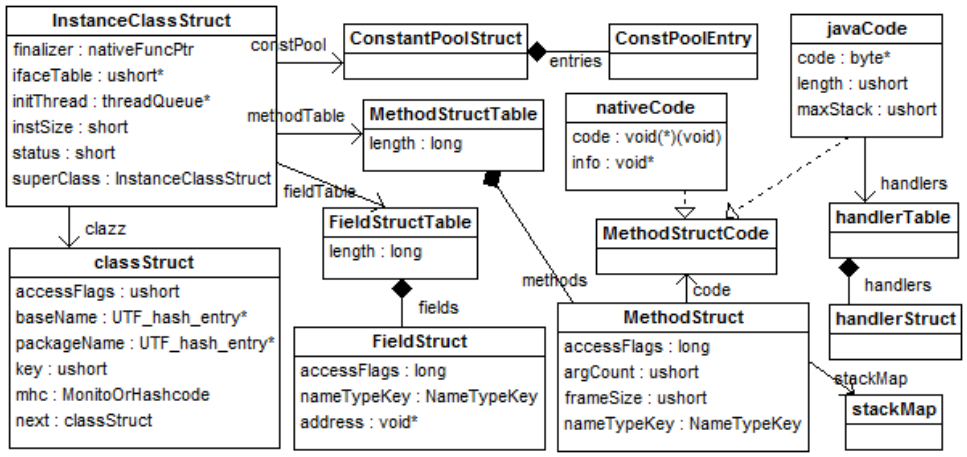


FIGURE 2 – Représentation des métadonnées Java dans KVM1.1.

possible. Cette démarche implique plusieurs choix comme point départ :

1. exclure de notre étude les JVM spécialement conçues pour les systèmes enfouis [7, 5, 14] car nous voulons d'une part observer de plus grosses applications Java qui sont impossibles à tester dans ce genre de machines virtuelles et d'autres part car nous voulons rester le plus près possible des JVM classiques ;
2. éviter de concevoir une nouvelle JVM qui est un processus long et fastidieux, mais plutôt s'assurer dans un premier temps que la modification d'une JVM classique suffit ;
3. ne faire aucune hypothèse concernant le cache. En d'autres termes, notre analyse ne porte pas sur l'évaluation d'une politique de cache, la manière dont les données sont récupérées de la mémoire externe (présence ou non d'un système de fichier...).

Pour réunir ces trois critères, nous avons donc choisi d'instrumenter KVM 1.1 de Sun [19]. Cette VM est la JVM de référence J2ME-CLDC. Elle est aussi très bien documentée et donc facile d'accès. Notre instrumentation a consisté à modifier le code source pour encapsuler le modèle de métadonnées proposé par KVM (Fig. 2) dans des macros C pour générer des traces d'accès et des cartes de mémoires (détaillées Section 5.2.2).

Nous avons également créé un simulateur d'architecture mémoires hiérarchiques pour observer finement la problématique des caches. À partir de traces comme celles que nous générons avec notre KVM instrumentée, nous sommes capable de réévaluer les accès mémoires d'un programme au travers de plusieurs critères comme différents niveaux de cache, différentes propriétés de cache (taille de pages, nombre de pages, politique de remplacement...), différents types de mémoire externe (NOR, NAND...) ainsi que leurs propriétés, notamment leur temps de latence comme décrit Section 2.4.

4.3. Les choix des programmes de tests

Il existe de nombreux programmes d'évaluation et de tests pour les machines virtuelle Java, y compris au niveau J2ME. Malheureusement, ces programme sont bien souvent conçus pour tester les aspects fondamentaux d'une JVM comme le nombre de bytecodes exécutés à la seconde et sont rarement riches en fonctionnalité objet. Un *benchmark* judicieux pour notre étude est le benchmark de Richards [17]. Le cœur de ce programme simule le gestionnaire de tâches d'un noyau de système d'exploitation. Sa version en Java est très intéressante et concrète car elle offre 7 versions différentes d'une même fonctionnalité, en balayant un large spectre des techniques de programmation en langage orienté objet [4], de plus basiques aux plus riches :

– V1 est une traduction directe du benchmark dans le style procédural du C sans fonctionalité *objet* ;
– V2 ajoute le mot clé *final* quand c'est possible pour réduire les coûts d'appel de méthodes Java ;
– V3 remplace le *switch/case* par des variables d'état ;
– V4 réécrit par héritage objet la notion de « tâche » mais en gardant tous les champs *public* ;
– V5 ajoute des accesseurs virtuels à ces champs, dans l'esprit de l'encapsulation objet ;

– V6 ajoute à 5 le mot clé *final* quand c'est possible pour réduire les coûts d'appel de méthodes Java ;
– V7 est une version proche du comportement d'un *framework* en utilisant des interfaces.
Chacune de ces versions va solliciter de plus en plus les métadonnées, ce qui va nous permettre de mesurer l'impact de l'utilisation d'un cache de métadonnées. Nous verrons par la suite que developper des applications Java pour nos cibles à la manière d'une application pour station (version 7) n'est pas anodin.

5. Résultats, observations et analyses

Cette Section propose une analyse des résultats obtenus par rapport aux objectifs fixés en Section 4.2 suivant trois axes : analyse des types de métadonnées (Section 5.1), analyse de la fréquence d'accès (Section 5.2.2), analyse des accès à travers un cache (Section 5.3).

5.1. Analyse par le type de métadonnée

5.1.1. Empreinte mémoire de KVM

KVM compilée pour un processeur ARM représente 131Ko. La taille par défaut réservée pour le tas est de 256Ko, ce qui donne une empreinte mémoire de 387Ko dans l'espace d'adressage du CPU. Les objets Java et les métadonnées sont tous alloués dans le tas. Une fois chargée par KVM, l'API CLDC1.1, à elle seule, occupe 130Ko, soit 51% de l'espace disponible. Les 7 versions du Richards representent quant à elle 48Ko de métadonnées. En suivant notre hypothèse de métadonnées externalisées, cela représente donc 178Ko, soit 70% d'espace libéré au sein du tas. On peut également prendre un autre point de vue et considérer que l'on peut plutôt réduire l'empreinte mémoire globale en réduisant le tas pour passer à 209Ko, soit un gain de 46%. Dans le cas des Richards, on peut même descendre à 151Ko (+60%) en réduisant le tas à 20Ko sans déclencher de *garbage collection*. Nous verrons dans la section perspective comment nous pourrions encore réduire cette taille pour atteindre les limites des capteurs réseau ou des cartes à puce (i.e 128Ko).

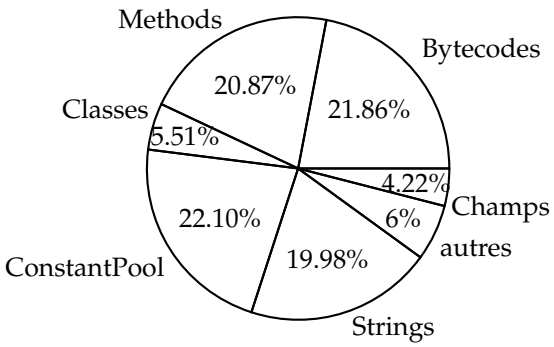


FIGURE 3 – RichardsV7 dans *SUN's KVM 1.1* - 133888 octets utilisés par les métadonnées : 126 classes chargées.

5.1.2. Proportion des différents types de métadonnées Java

La Figure 3 présente un exemple de distribution entre les différents types de métadonnées issus du modèle proposé par KVM. Les proportions présentés ici sont à peu près les même pour toutes les applications que nous avons étudiées. On notera d'abord que la part des *bytecodes* n'est que d'environ 22%. Alors que l'ensemble {classes, méthodes, champs} représente un tiers des métadonnées. On note enfin que le *constant pool* représente près de 22% de l'espace utilisé et les travaux sur son optimisation étaient justifié [14, 17, 7]. « autres »rassemble des métadonnées plus spécifiques comme les « Tables de hachage » interne à KVM pour organiser le stockage des classes, les gestionnaires d'exceptions, ou des informations de contrôle de pile propres au format J2ME-CLDC.

5.2. Analyse de la fréquence d'accès

5.2.1. Analyse de la fréquence d'accès par types de métadonnées

Notre première observation majeure s'appuie sur les résultats fournis par le Tableau 1 contenant une compilation des données collectées par notre KVM instrumentée lors de l'exécution des sept versions du benchmark Richards. La deuxième colonne rapporte la fréquence d'accès aux métadonnées de type bytecode, la troisième pour le type méthode, la quatrième pour les autres métadonnées. Les cinquième et sixième colonnes donne respectivement le volume de métadonnées lues et écrites. Enfin, la dernière colonne représente la moyenne du nombre de métadonnées lues (en octets) pour chaque bytecode executé.

Richards Version	% accès aux bytecode	% accès aux methodes	% accès autres	métadonnées lues	métadonnées écrites	métadonnées lues / bytecode
1	65.26	25.83	8.91	284 Mo	137 Ko	5.5
2	73.46	17.13	9.41	217 Mo	137 Ko	3.9
3	59.40	32.02	8.58	417 Mo	137 Ko	6.9
4	40.69	49.83	9.48	928 Mo	141 Ko	12.5
5	18.67	72.96	8.37	3836 Mo	145 Ko	33.6
6	73.46	17.87	8.67	991 Mo	145 Ko	8.7
7	18.96	72.68	8.36	3844 Mo	145 Ko	33.7

TABLE 1 – Évaluation des accès aux métadonnées des benchmark Richards dans KVM 1.1.

Observation 1 *Le volume d'écriture est effectivement anecdotique par rapport aux lectures.*

Observation 2 *Dans certaines versions du Richards (2 et 6), KVM accède plus de 70% du temps à des métadonnées de type « bytecode ». Au contraire, dans d'autres versions (5 et 7), KVM accède plus de 70% du temps à des métadonnées de type « méthode ».*

Observation 3 *La façon dont une application Java est développée influence le nombre d'accès aux métadonnées. En moyenne, l'execution d'un bytecode dans une application écrite par héritage, surcharge et encapsulation déclenche 10 fois plus d'accès aux métadonnées.*

Ces deux dernières observations mettent en évidence un premier point essentiel de notre étude. L'externalisation des métadonnées ne peut pas se faire au détriment de ce qui fait la force des langages orienté objet : l'héritage, la surcharge, et l'encapsulation. Il est évident que dans cette perspective, l'utilisation d'un cache sera la resource critique pour les performances des applications. De plus, il apparait clairement dans nos résultats qu'une attention particulière doit être portée sur la métadonnée de type méthode.
Il nous faut donc regarder plus finement les accès afin de savoir si nous sommes réellement face à un problème rédhibitoire à l'externalisation des métadonnées.

5.2.2. Analyse de la fréquence d'accès octets par octets

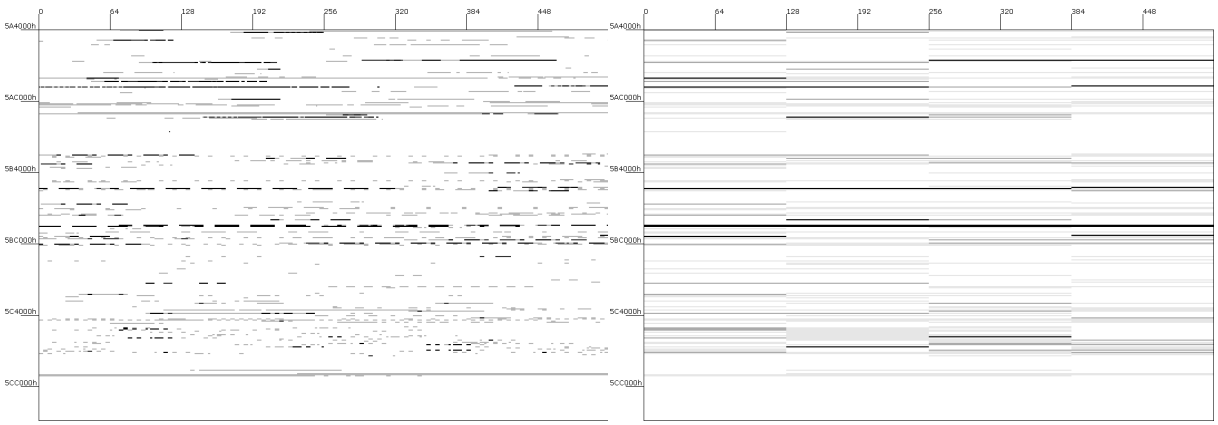
Une cartographie des accès mémoires permet d'avoir une vision plus fine des fréquences d'accès en regardant chaque octet (Figure 4a). Elle peut être représentée sous forme de carte de chaleur mettant en évidence les points chauds, accédés très fréquemment (en noir sur la figure), les points froids peu accédés (en gris), et les blancs inutilisés.
Si un point chaud est défini par une certaine fréquence d'accès, nous devons alors déterminer un seuil au-delà duquel, un octet lu en mémoire externe est chaud ou froid. Il existe deux approches possibles dans le cas d'une future utilisation d'un cache :

- une approche basée sur la recherche de performance : identifier tous les octets nécessaires pour atteindre un certain taux de présence en cache (Hit ratio). *Ex : tous les octets représentants à eux seuls 99% des accès mémoires.* ;
- une approche basée sur la limite en espace mémoire, comme une taille de cache maximale. *Ex : les 1024 plus chauds octets.*

L'identification de points chauds est prometteuse sur le papier. En effet, si nous pouvions les connaître à l'avance, ceux-ci pourraient être placés dès le début de l'exécution de la VM dans un espace mémoire privilégié. Si cette stratégie est non triviale à mettre en œuvre concrètement, d'un point de vue analytique, elle permet deux nouvelles observations, la première basée sur la performance, la seconde sur l'espace mémoire. Voici ce que nous avons pu constater :

Observation 4 *986 octets (9,5% des métadonnées utilisées) représentent 99,5% des accès mémoires aux métadonnées dans la version 1 du Richards, alors que 2211 octets (17,5% des métadonnées utilisées) sont nécessaires dans le version 7 pour atteindre 99,5%.*

Observation 5 *1024 octets représentent 99,96% des accès mémoires aux métadonnées dans la version 1 du Richards, alors que dans la version 7, ils ne représentent plus que 89,00% des accès.*



(a) Carte des accès aux métadonnées octet par octet

(b) Carte des accès aux métadonnées à travers des pages de 128 octets

FIGURE 4 – Phénomène de dilution spatiale des points chauds (Benchmark Richards V7).

Il apparait ici clairement qu'un paramètre primordiale du cache sera sa taille. Celui-ci doit en effet être suffisement large pour contenir un maximum de point chaud. Néanmoins, les résultats sont encourageants. Si 2211 octets représentent 17,5% des métadonnées réellement utilisées, un cache de 3Ko en mémoire interne ne représente que 1,7% de l'espace libéré dans le tas grâce au stockage des métadonnées dans une mémoire secondaire, ce qui laisse encore un peu de marge.

5.3. Points chauds dans le contexte d'un cache

Notre dernier axe d'analyse porte sur l'hypothèse que nous formulons d'utiliser des mémoires externes devant être accédées à travers un cache.

5.3.1. Introduction à la problématique apportée par un cache

La Figure 4a nous donne deux informations supplémentaires sur les métadonnées.

Observation 6 *Les points chauds sont très petits.*

Les points chauds sont rarement adjacents. Cette fragmentation est introduite par les structures de données C. Deux champs consécutifs d'une même structure ne sont pas sollicités avec la même intensité, ce qui provoque des « trous » qui, comme nous allons le voir, posent des problèmes.

Observation 7 *Les points chauds sont répartis inégalement et sur une large plage.*

Ces observations valent également pour les données plus froides. Dans le contexte de la pagination à la demande, ces remarques ont une conséquence directe sur la pertinence du contenu du cache. Imaginons que le système veuille accéder à une métadonnées Δ. Si Δ n'est pas trouvée en cache (on parle ici de défaut de page), le gestionnaire de cache doit lire Δ dans la mémoire externe. Mais le cache en récupérant Δ, récupère également les données adjacentes, présentes dans la même page. À cet instant, on ne peut pas savoir si les autres données contenues dans la page éviteront ou non d'autres défauts de page. On peut seulement émettre deux hypothèses, contradictoires, que nous allons vérifier et surtout évaluer dans cette Section :

– hypothèse 1 : plus une page est grosse, plus on augmente les chances de récupérer un grand nombre de points chauds en un seul accès à la mémoire externe ;
– hypothèse 2 : plus une page est petite, plus on diminue le risque d'amener en cache des données blanches, inutiles qui immobilisent de l'espace-cache pour rien (on parle ici de pollution de cache).

Le Tableau 2 livre les perspectives de ces deux hypothèses. Il s'agit de résultats obtenus à partir de traces du benchmark Richards dans le contexte où une Flash NAND serait virtualisée par un cache de 2048 octets. Pour chaque colonne de ce tableau, nous varions la taille d'une page de cache, ainsi que leur

Richards Version	4 pages de 512o	8 pages de 256o	16 pages de 128o	32 pages de 64o	64 pages de 32o	128 pages de 16o	256 pages de 8o	Flash NOR
1	0,33	0,98	2,32	3,76	20,48	23,77	23,78	23,84
7	0,44	0,88	1,85	2,62	3,35	4,01	4,07	23,84

TABLE 2 – Débits obtenus par simulation et donnés en Mo/s d'un cache de 2048 octets dans le cas où la mémoire externe est de la flash NAND, comparés au débit d'une flash NOR interne.

nombre pour garder une taille de cache équivalente. Nous avons ici simulé une Flash NAND typique comme décrit Section 2.4. La dernière colonne remplace le couple cache/NAND par de la Flash NOR aux propriétés vues Section 2.3. Le cache quant à lui se situe en RAM et est géré par la politique de remplacement de page LRU (Least Recently Used). Les résultats sont exprimés en terme de débit de lecture (en Mo/s), pour pouvoir comparer des architectures avec cache et des architectures sans cache.

Observation 8 *Plus les pages sont petites et leur nombre grand, meilleures sont les performances.*

Le premier facteur d'amélioration du débit est l'augmentation du nombre de pages disponibles en cache. Ce facteur permet à la politique de remplacement de page d'imposer au cache une meilleure gestion de son contenu dans le temps. Ce paramètre a déjà été parfaitement décrit dans la littérature sur les caches [18]. Il est évident que 256 pages de 512 octets (128Ko) seront plus performantes que 256 pages de 8 octets (2Ko), mais cette solution confortable est impossible si la cache ne peut pas excéder, par exemple, 2Ko d'occupation de RAM. Néanmoins, dans le tableau 2, on constate que 256 pages de 8 octets sont déjà beaucoup plus performantes de 4 pages de 512. C'est donc que la taille d'une page a également son importance dans le cadre fixé par nos hypothèses.

5.3.2. Phénomène de dilution spatiale

Lorsque l'on compare les figures 4a et 4b, on remarque que certains points chauds disparaissent sous l'effet du regroupement par pages des données. Dans le même temps et sous le même effet, beaucoup d'autres régions mémoire se réchauffent, tout en englobant un nombre significatif de données blanches. C'est ce que nous appellerons le phénomène de dilution spatiale. La chaleur globale d'une page de données est lissée à la moyenne de la chaleur des octets qui la compose. La dilution spatiale n'est pas un critère d'optimisation future mais elle met en évidence la difficulté du cache à garder le plus longtemps possibles les données les plus sollicités lorsqu'il doit gérer des grosses pages. Le corolaire à la dilution spatiale est le taux de concentration d'une page en points chauds. Plus ce taux est élevé pour une page, plus cette page s'avère pertinente à garder en cache.

Tout cela signifie que réduire la taille d'une page diminue la dilution spatiale. De plus, comme les points chauds sont rarement adjacents du fait des structure de données correspondantes, la taille optimale d'une page de cache devient très proche de la granularité d'une métadonnée simple, (*i.e.* 8 à 32bits). Il y a alors un effet de seuil où le cache va finalement pouvoir garder de plus en plus de données parmi les plus fréquentes, en ne s'encombrant quasiment jamais de données blanches. Cet effet de seuil commence à apparaitre avec 64 pages de 32 octets pour la version 1 du benchmark Richards. Dès ce stade, et pour ce programme, la dilution spatiale devient suffisamment négligeable pour que le cache puisque garder les 986 octets (et leur données adjacentes) identifiés en Section 5.2.2, mais tout en continuant à avoir suffisamment de pages disponibles pour ne pas générer trop de défaut de pages pour les autres données, moins fréquentes, mais qui devront être rapatriées en cache à un moment ou à un autre.

On remarquera que dans le cas de la version 7 du benchmarks Richards, cet effet de seuil n'est jamais atteint et que les performances ne décollent pas. En effet, un cache de 2Ko n'est pas suffisant pour ce programme pour atteindre le point de convergence entre taille de page, nombre de pages et quantité de points chauds.

5.4. Synthèse de ces observations

Notre analyse pointe deux observations fondamentales qui doivent guider les travaux futurs. La première est la façon dont les applications Java sont développées. Nous avons noté que le principal point de contention dans l'accès aux métadonnées est l'accès aux méthodes virtuelles, phénomène amplifié par des techniques de programmation très répandues. Il semble donc essentiel d'optimiser ces accès, ce

qui ne peut pas se faire *a priori* en pensant la mise en œuvre de table de méthodes virtuelles indépendamment du fait qu'elles seront sollicitées à travers un cache.
La seconde observation porte sur le modèle qui structure les métadonnées. Ce modèle décrit la façon dont celles-ci sont réparties et organisées dans la mémoire. Nous avons ainsi pointé deux conséquences qui sont la fragmentation des points chauds et le phénomène de dilution spatiale. Il semble donc essentiel d'optimiser la manière dont les méta-données sont organisées. Une attention particulière doit être portée quant au modèle qui les décrit (et par extension aux structures de données C correspondantes).
Néanmoins, ces deux points ne sont que des imperfections dûes à l'utilisation d'un cache et ne font pas obstacle à la faisabilité. Au contraire, nous avons montré que quelques paramètres de cache bien adapté peuvent facilement et efficacement remédier à ces carences.

6. Perspectives

Le stockage des métadonnées Java dans une mémoire secondaire ne résout pas tout les problèmes de contraintes mémoires bridant l'utilisation d'une machine virtuelle dans un système enfoui. Une machine virtuelle Java ouverte sur l'extérieur et permettant le chargement *a posterio* de code, comme KVM ou Squawk, sont encore tros grosses pour tenir dans les cibles auxquelles JavaCard ou Darjeeling répondent. Toutefois ces projets incluent des idées qui pourraient nous permettre d'atteindre notre objectif.
JITS et Squawk ont montré qu'il était tout à fait envisageable d'écrire le chargeur de classe en Java. Appliquer ce concept à KVM libérerait 40Ko dans la mémoire interne, ce qui permettrait à KVM de tenir dans moins de 128Ko. Écrire un chargeur de classes en Java peut le rendre plus complexe et plus gros que celui natif d'origine, mais son utilisation n'est que ponctuelle et peut, grâce à notre solution, être stocké en mémoire externe sans occuper inutilement de l'espace. D'autres composants comme les pilotes de périphérique, les systèmes de fichiers, la couche réseau peuvent eux aussi être en grande partie écrits en Java et ne pas réquérir de système d'exploitation sous-jacent.
De plus, il est toujours envisageable d'optimiser hors-ligne le code Java et par exemple essayer d'augmenter sa localité spatiale en fusionnant des méthodes (*i.e. inlining*) pour réduire la fréquence d'accès aux métadonnées de type méthode.

7. Conclusion

L'utilisation de langages de haut niveau, basés sur des machines virtuelles, comme le Java, est encore aujourd'hui un challenge pour les systèmes fortement restreints en espace mémoire. Si une application Java est légère seule, elle nécessite une infrastructure logicielle sous-jacente encore aujourd'hui trop imposante pour embarquer toutes les fonctionnalités du Java standard. Pour résoudre ce problème, nous imaginons une solution qui consiste à déporter un maximum de ce qui peut l'être dans une mémoire de stockage secondaire. Ce papier est une étude préliminaire à cette idée. Nous avons identifié en Section 4.2 que, dans ce contexte, les métadonnées Java étaient le meilleur groupe de données à placer hors de l'espace de stockage de code classique. Nous avons également fourni Section 5 une analyse détaillée du comportement des métadonnées Java à l'exécution suivant trois axes : le type de métadonnées, la fréquence d'accès et les effets de l'utilisation d'un cache.
Nous avons constaté que trois facteurs majeurs viennent pour l'instant perturber les performances : le haut niveau d'abstraction dans la programmation d'applications, la fragmentation des métadonnées et la dilution spatiale occasionnée par le cache. Le premier est le plus complexe car il représente à lui seul l'intérêt d'utiliser le Java, par exemple, dans les systèmes enfouis. Cependant, ce problème doit pouvoir être corrigé à la source en créant une JVM répondant à l'exigence d'un cache, avec une gestion des méthodes virtuelles adaptée à cette exigence. Par exemple, un cache asynchrone permettrait un meilleur recouvrement des opérations lentes que sont les défaut de page. Le deuxième facteur de ralentissement est la conséquence là aussi du *design* d'une JVM qui n'est pas pensée dès l'origine pour ce genre d'utilisation. Nous avons montré que le troisième facteur, qui est le cœur de notre idée, pouvait être atténué par l'ajustement des tailles de pages de cache, mais sans pouvoir proposer de solution idéale pour l'instant, car nous pensons, au vue de nos analyses, qu'une solution ne peut pas être idéale sans être une réponse globale aux trois facteurs identifiés.

Bibliographie

1. H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. Mantis : system support for multimodal networks of in-situ sensors. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, WSNA '03. ACM, 2003.
2. Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Pearson, 4th edition, 2005.
3. Faisal Aslam, Luminous Fennell, Christian Schindelhauer, Peter Thiemann, Gidon Ernst, Elmar Haussmann, Stefan Rührup, and Zastash Uzmi. Optimized java binary and virtual machine for tiny motes. In *Distributed Computing in Sensor Systems*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010.
4. Joshua Bloch. *Effective Java : a programming language guide ; 2nd ed.* Addison-Wesley, 2008.
5. Niels Brouwers, Koen Langendoen, and Peter Corke. Darjeeling, a feature-rich vm for the resource poor. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09. ACM, 2009.
6. A. Caracas, T. Kramp, M. Baentsch, M. Oestreicher, T. Eirich, and I. Romanov. Mote runner : A multi-language virtual machine for small embedded devices. *Sensor Technologies and Applications, International Conference on*, pages 117–125, 2009.
7. Alexandre Courbot, Gilles Grimaud, and Jean-Jacques Vandewalle. Efficient off-board deployment and customization of virtual machine-based embedded systems. *ACM Trans. Embed. Comput. Syst.*, 9 :21 :1–21 :53, 2010.
8. Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04. IEEE Computer Society, 2004.
9. Lin Gu and John A. Stankovic. t-kernel : providing reliable os support to wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06. ACM, 2006.
10. Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *SIGOPS Oper. Syst. Rev.*, 34 :93–104, November 2000.
11. Andreas Lachenmann, Pedro José Marrón, Matthias Gauger, Daniel Minder, Olga Saukh, and Kurt Rothermel. Removing the memory limitations of sensor networks with flash-based virtual memory. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07. ACM, 2007.
12. Philip Levis and David Culler. Maté : a tiny virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.*, 36 :85–95, 2002.
13. Tim Lindholm and Frank Yellin. *Java(TM) Virtual Machine Specification, The (2nd Edition)*. Prentice Hall PTR, 2 edition, 1999.
14. Oracle Technology Network. Java card platform specification. In *http ://www.oracle.com/technetwork/java/javacard/*.
15. Chanik Park, Junghee Lim, Kiwon Kwon, Jaejin Lee, and Sang Lyul Min. Compiler-assisted demand paging for embedded systems with flash memory. In *Proceedings of the 4th ACM international conference on Embedded software*, EMSOFT '04. ACM, 2004.
16. Ulrik Pagh Schultz, Kim Burgaard, Flemming Gram Christensen, and Jørgen Lindskov Knudsen. Compiling java for low-end embedded systems. *SIGPLAN Not.*, 38 :42–50, 2003.
17. Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. Javatm on the bare metal of wireless sensor devices : the squawk java virtual machine. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06. ACM, 2006.
18. Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14 :473–530, September 1982.
19. A. Taivalsaari, B. Bill, , and D. Simon. The spotless system : Implementing a java system for the palm connected organizer. *Technical Report SMLI TR-99-73, Sun Microsystems Research Laboratories, Mountain View*, 1999.